CS 102
Computer Architecture

Lecture 2: *Introduction to C, Part I*

Instructor:
Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

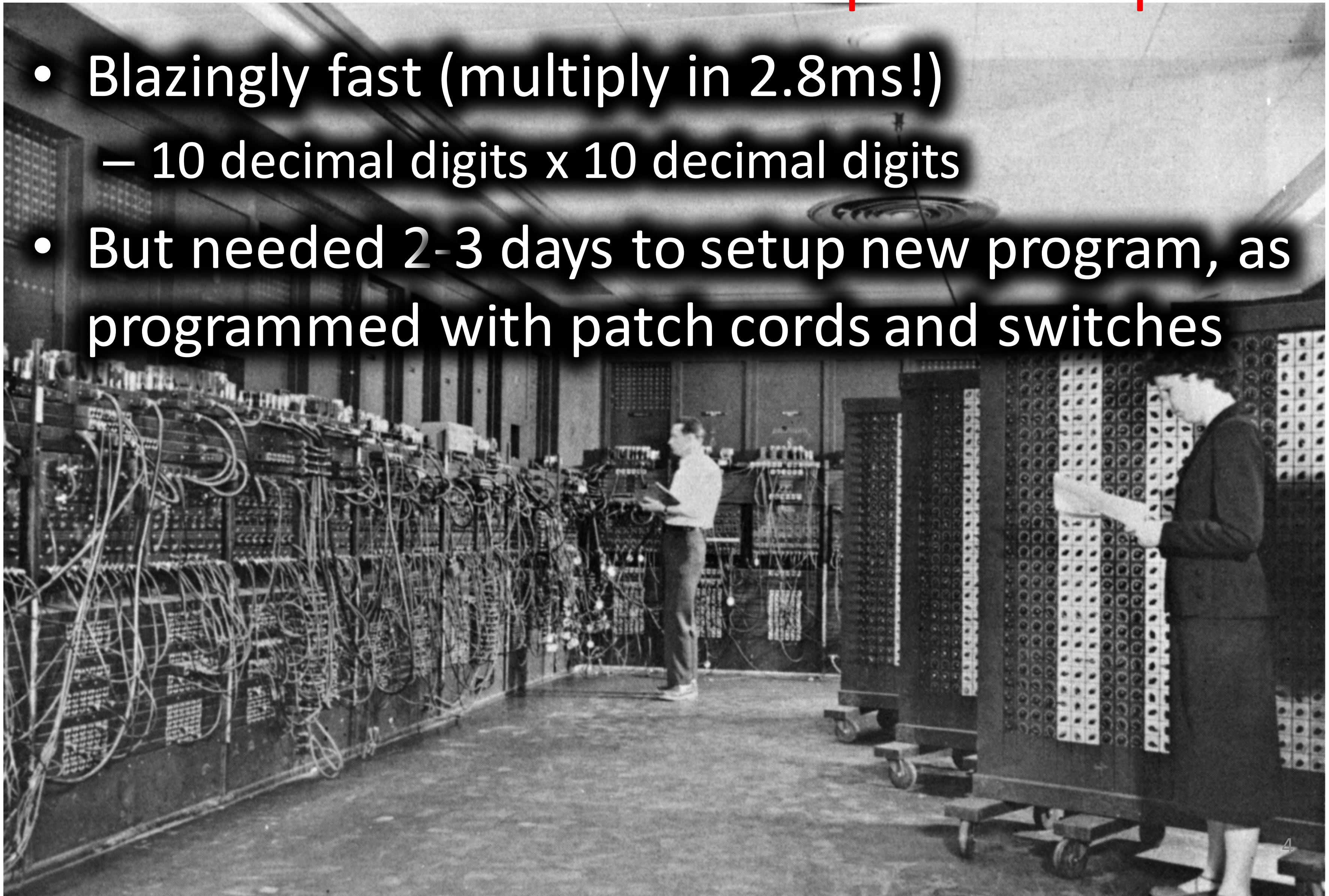
Agenda

- Compile vs. Interpret
- Administrivia
- Quick Start Introduction to C
- News/Technology Break
- Pointers
- And in Conclusion, …

Agenda

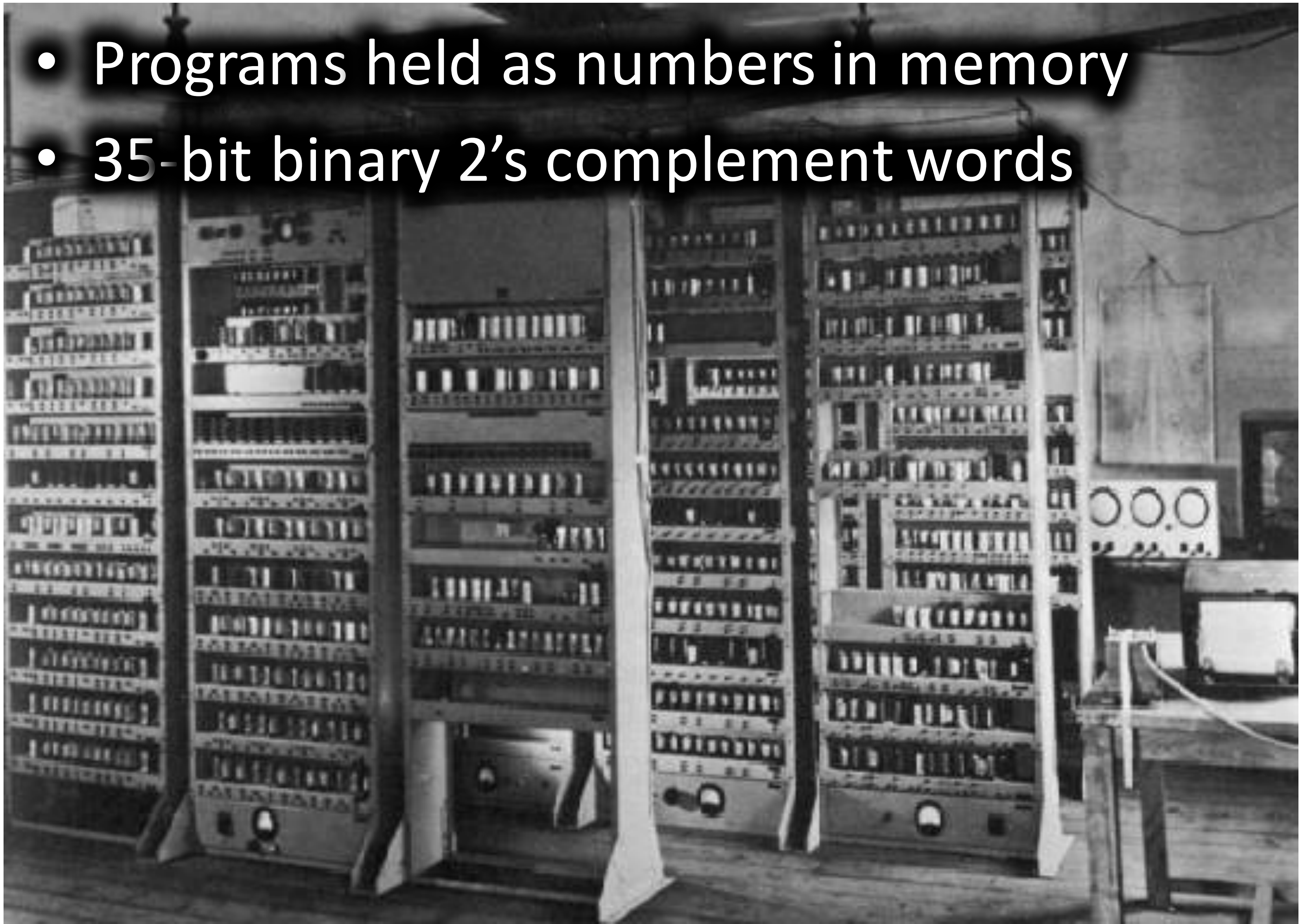
- Compile vs. Interpret
- C vs. Java vs. Python
- Administrivia
- Quick Start Introduction to C
- News/Technology Break
- Pointers
- And in Conclusion, …

ENIAC (U.Penn., 1946)
First Electronic General-Purpose Computer



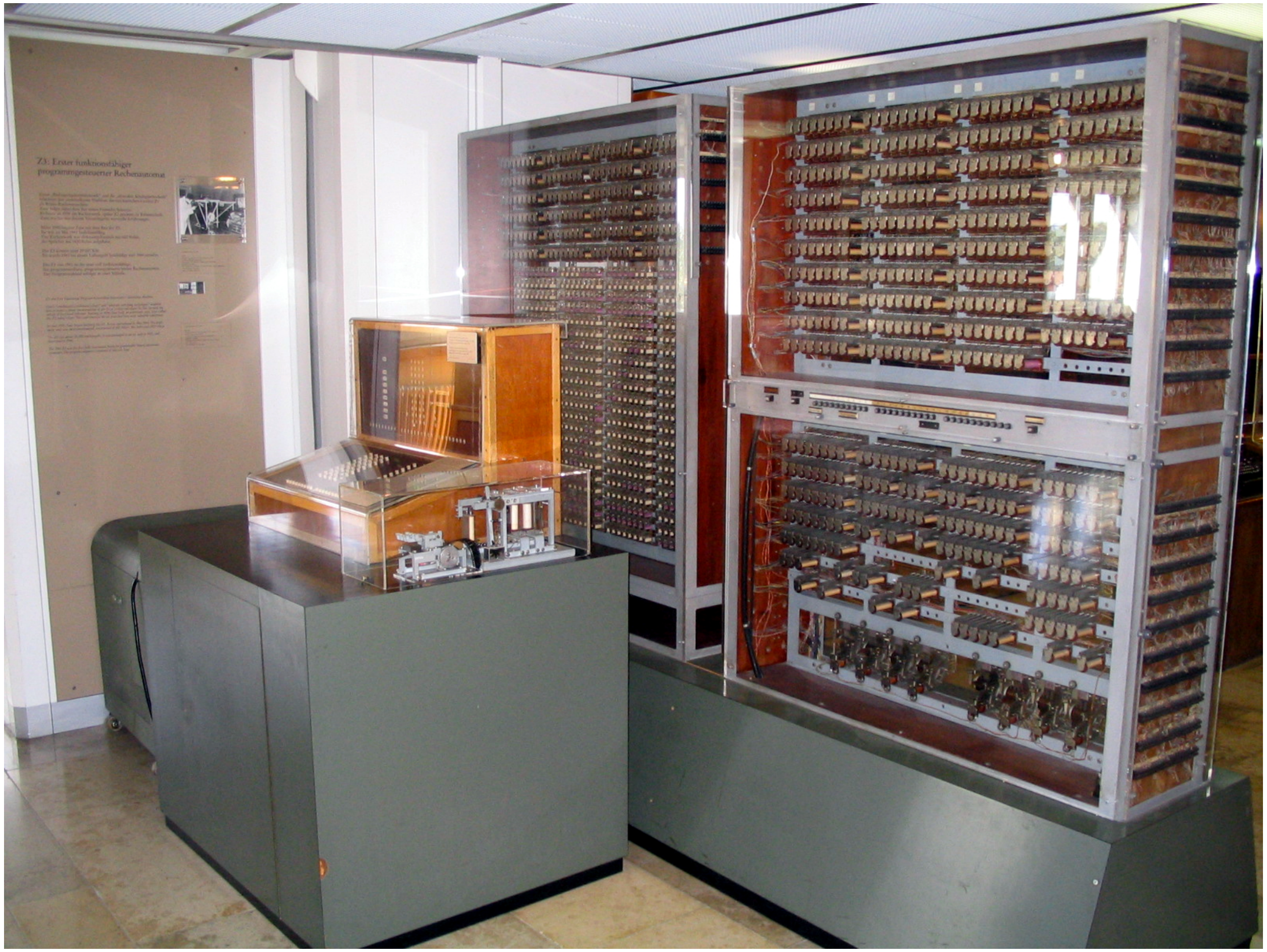
- Blazingly fast (multiply in 2.8ms!)
  - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches

EDSAC (Cambridge, 1949)
First General Stored-Program Computer

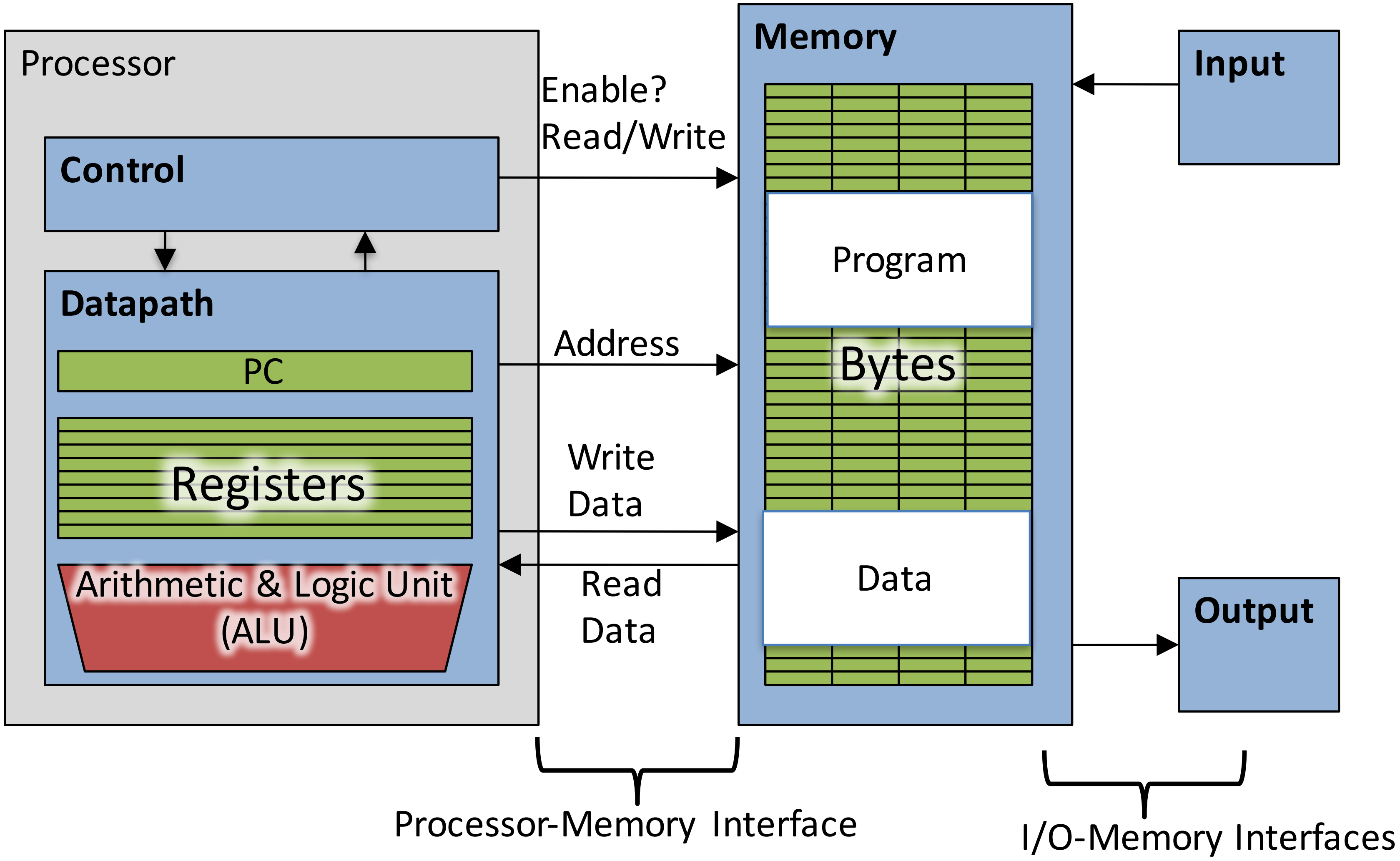


- Programs held as numbers in memory
- 35-bit binary 2's complement words

But actually: first working programmable, fully automatic digital computer: Zuse Z3 (Germany 1941)



Components of a Computer



Great Idea: Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

We are here!

Compiler

Assembly Language Program (e.g., MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

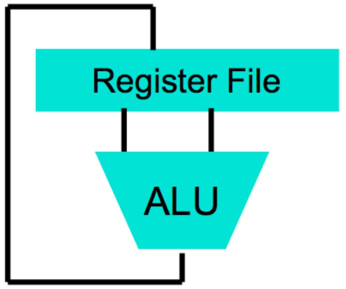
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

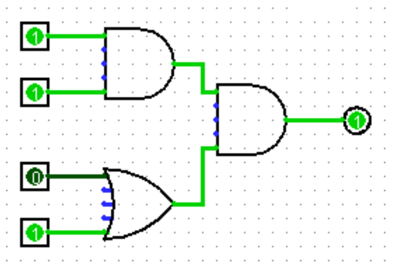
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

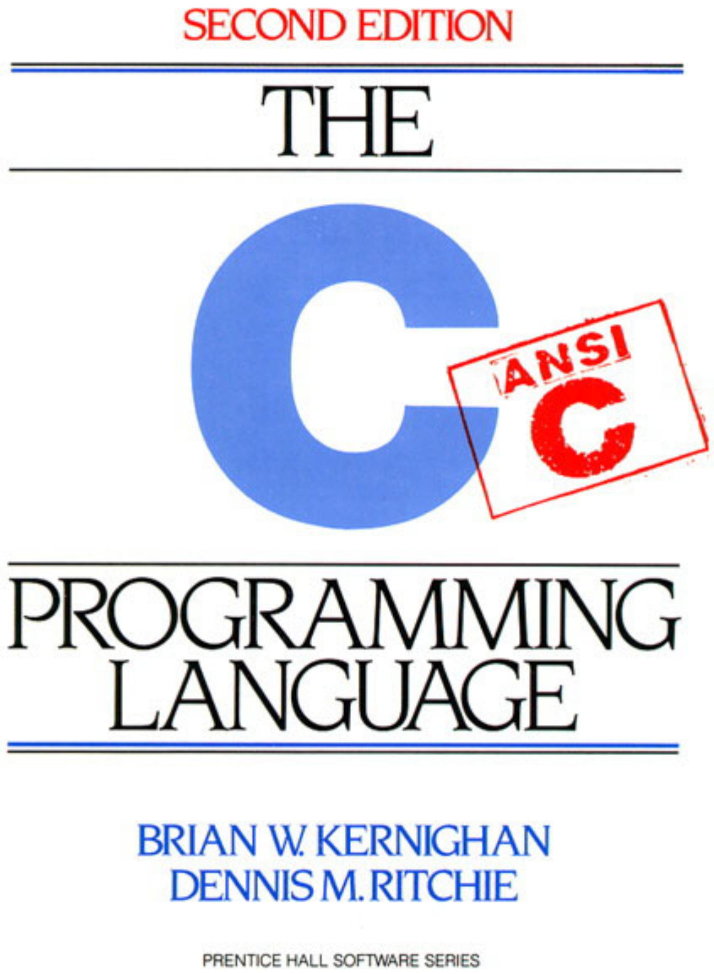


Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to C
"The Universal Assembly Language"



Intro to C

- *C is not a "very high-level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.*

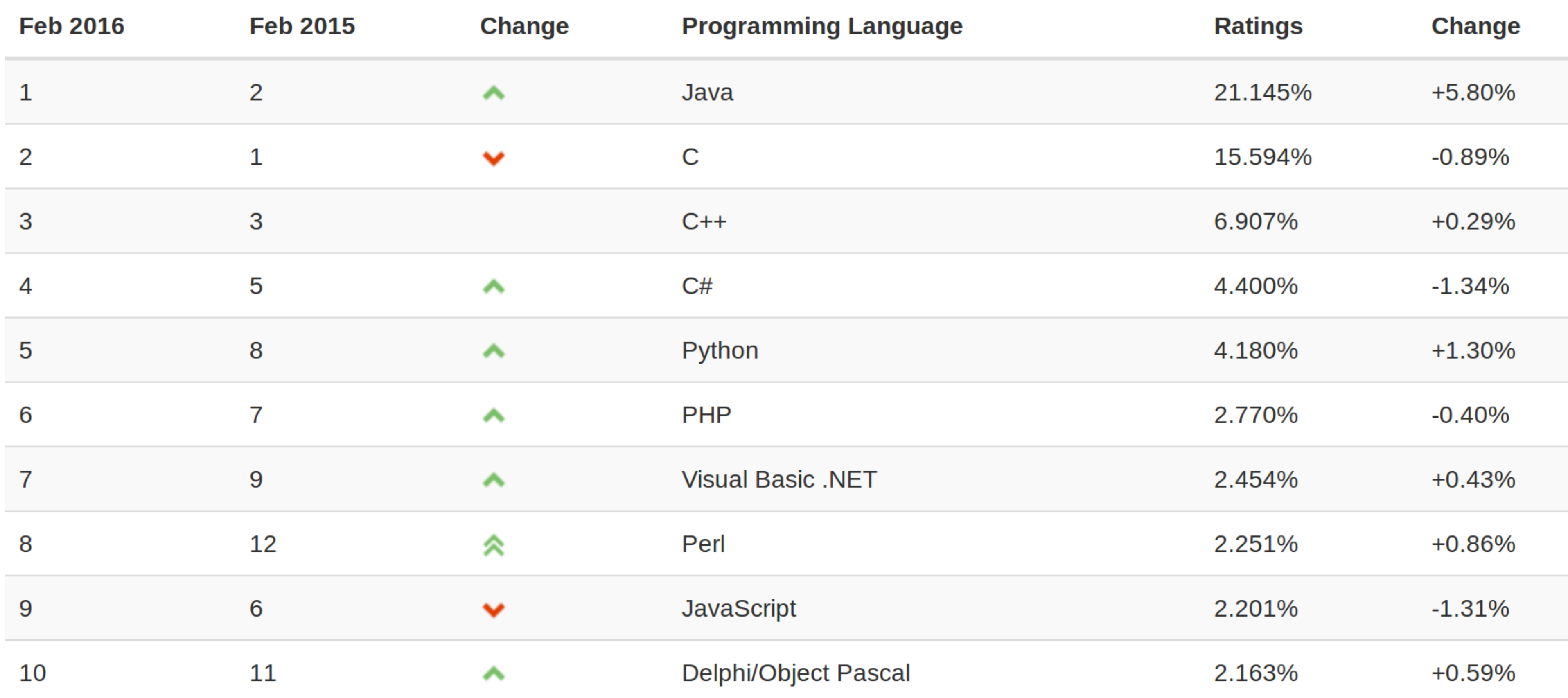
  – Kernighan and Ritchie

- Enabled first operating system not written in assembly language: *UNIX* - A portable OS!

Intro to C

- Why C?: *we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism*
- C and derivatives (C++/Obj-C/C#) still one of the most popular application programming languages after >40 years!

TIOBE Index of Language Popularity

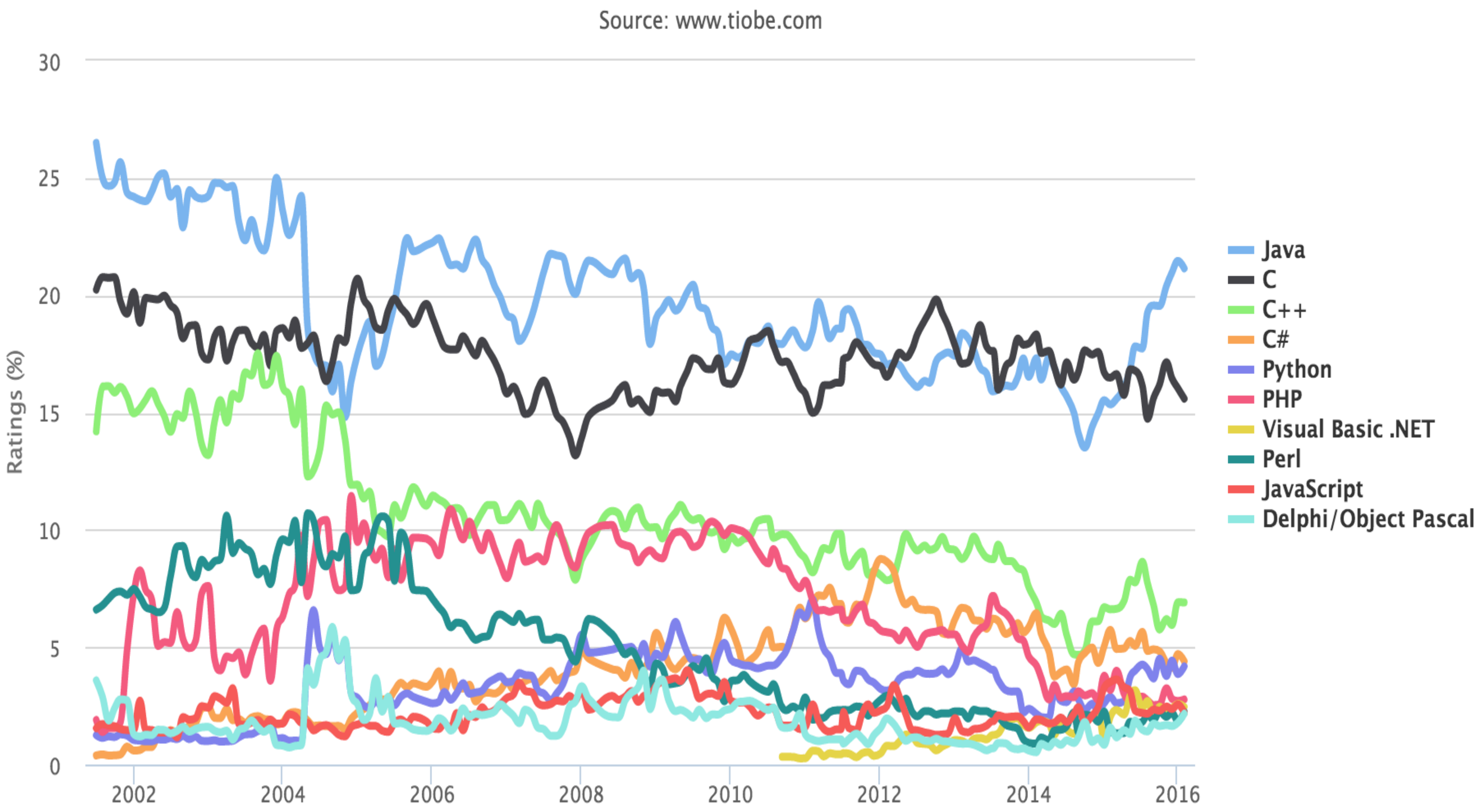


Feb 2016	Feb 2015	Change	Programming Language	Ratings	Change
1	2	▲	Java	21.145%	+5.80%
2	1	▼	C	15.594%	-0.89%
3	3		C++	6.907%	+0.29%
4	5	▲	C#	4.400%	-1.34%
5	8	▲	Python	4.180%	+1.30%
6	7	▲	PHP	2.770%	-0.40%
7	9	▲	Visual Basic .NET	2.454%	+0.43%
8	12	▲▲	Perl	2.251%	+0.86%
9	6	▼	JavaScript	2.201%	-1.31%
10	11	▲	Delphi/Object Pascal	2.163%	+0.59%

The ratings are based on the number of skilled engineers world-wide, courses and third party vendors.

http://www.tiobe.com

TIOBE Programming Community Index



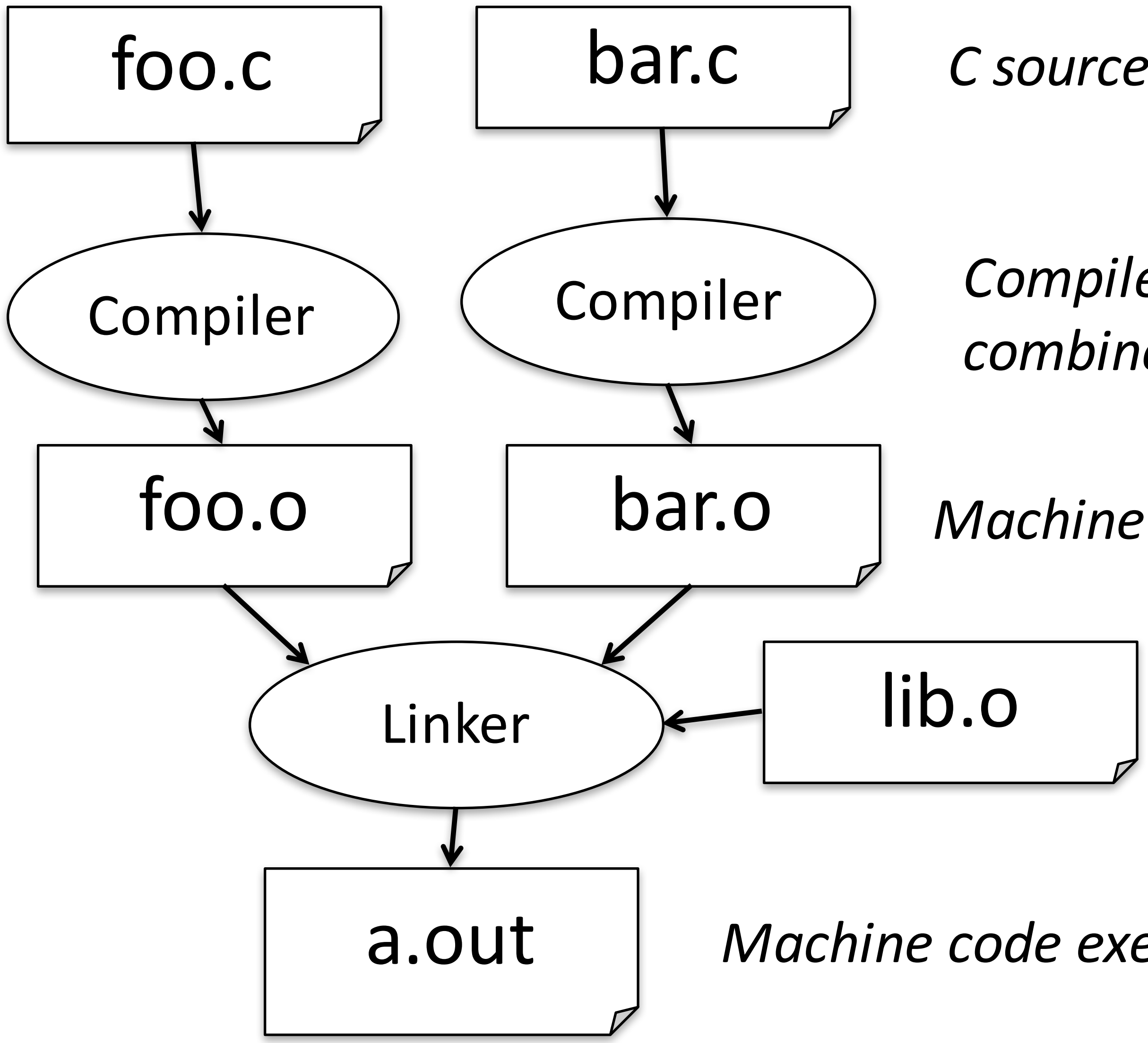
Disclaimer

- You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course
  - K&R is a must-have
    - Check online for more sources
- Key C concepts: Pointers, Arrays, Implications for Memory management
- We will use ANSI C89 – original "old school" C

Compilation: Overview

- C *compilers* map C programs into architecture-specific machine code (string of 1s and 0s)
  - Unlike *Java*, which converts to architecture-independent *bytecode*
  - Unlike *Python* environments, which *interpret* the code
  - These differ mainly in exactly when your program is converted to low-level machine instructions (“levels of interpretation”)
  - For C, generally a two part process of compiling .c files to .o files, then linking the .o files into executables;
  - Assembling is also done (but is hidden, i.e., done automatically, by default); we’ll talk about that later

C Compilation Simplified Overview (more later in course)



C source files (text)

Compiler/assembler combined here

Machine code object files

Pre-built object file libraries

Machine code executable file

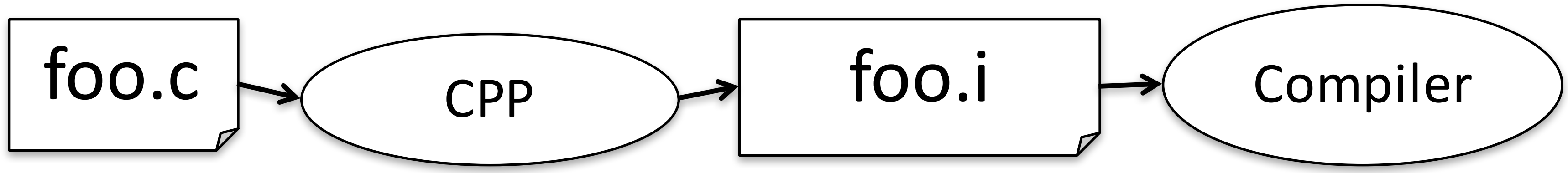
Compilation: Advantages

- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- Reasonable compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., MIPS vs. RISC-V) and the operating system (e.g., Windows vs. Linux)
- Executable must be rebuilt on each new system
  - I.e., “porting your code” to a new architecture
- “Change → Compile → Run [repeat]” iteration cycle can be slow during development
  - but Make tool only rebuilds changed pieces, and can do compiles in parallel (linker is sequential though -> Amdahl’s Law)

C Pre-Processor (CPP)



- C source files first pass through macro processor, CPP, before compiler sees code
- CPP replaces comments with a single space
- CPP commands begin with "#"
- #include "file.h" /* Inserts file.h into output */
- #include <stdio.h> /* Looks for file in standard location */
- #define M_PI (3.14159) /* Define constant */
- #if/#endif /* Conditional inclusion of text */
- Use –save-temps option to gcc to see result of preprocessing
- Full documentation at: **`http://gcc.gnu.org/onlinedocs/cpp/`**

Typed Variables in C

```
int   variable1   = 2;
float variable2   = 1.618;
char  variable3   = 'A';
```

- Must declare the type of data a variable will hold
  - Types can't change

Type	Description	Examples
int	integer numbers, including negatives	0, 78, -1400
unsigned int	integer numbers (no negatives)	0, 46, 900
float	floating point decimal numbers	0.0, 1.618, -1.4
char	single text character or symbol	'a', 'D', '?’
double	greater precision/big FP number	10E100
long	larger signed integer	6,000,000,000

Integers: Python vs. Java vs. C

Language	sizeof(int)
Python	>=32 bits (plain ints), infinite (long ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

- C: `int` should be integer type that target processor works with most efficiently
- Only guarantee: sizeof(`long long`) ≥ sizeof(`long`) ≥ sizeof(`int`) ≥ sizeof(`short`)
  - Also, `short` >= 16 bits, `long` >= 32 bits
  - All could be 64 bits

Consts and Enums in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float golden_ratio = 1.618;
const int days_in_week = 7;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants. Ex:

```
    enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
    enum color {RED, GREEN, BLUE};
```

Compare “`#define PI 3.14`” and “`const float pi=3.14`” – which is true?

A: Constants “PI” and “pi” have same type

B: Can assign to “PI” but not “pi”

C: Code runs at same speed using “PI” or “pi”

D: “pi” takes more memory space than “PI”

E: Both behave the same in all situations

Agenda

- Compile vs. Interpret
- C vs. Java vs. Python
- Administrivia
- Quick Start Introduction to C
- News/Technology Break
- Pointers
- And in Conclusion, …

Administrivia

- Find a partner for the lab and send your selection to Xu Qingwen (xuqw)
- Labs start next week! Check your schedule! You cannot get checked without a partner!
- The tasks for Lab 1 will be posted on the website today. Prepare for it over the weekend.
- HW1 has been posted. Ask questions about it on piazza. Or get help during the lab or during OH.

Agenda

- Compile vs. Interpret
- C vs. Java vs. Python
- Administrivia
- Quick Start Introduction to C
- News/Technology Break
- Pointers
- And in Conclusion, …

Typed Functions in C

```
int number_of_people ()
{
  return 3;
}

float dollars_and_cents ()
{
  return 10.33;
}

int sum ( int x, int y)
{
   return x + y;
}
```

- You have to *declare* the type of data you plan to return from a function
- Return type can be any C variable type, and is placed to the left of the function name
- You can also specify the return type as **`void`**
  - Just think of this as saying that no value will be returned
- Also necessary to declare types for values passed into a function
- Variables and functions MUST be declared before they are used

Structs in C

- Structs are structured groups of variables, e.g.,

```
typedef struct {
  int length_in_seconds;
  int year_recorded;
} Song;

Song song1;

song1.length_in_seconds =  213;
song1.year_recorded      = 1994;

Song song2;

song2.length_in_seconds =  248;
song2.year_recorded      = 1988;
```

Dot notation: **`x.y = value`**

A First C Program: Hello World

```
Original C:

main()
{
   printf("\nHello World\n");
}
```

```
ANSI Standard C:

#include <stdio.h>

int main(void)
{
   printf("\nHello World\n");
   return 0;
}
```

C Syntax: `main`

- When C program starts
  - C executable a.out is loaded into memory by operating system (OS)
  - OS sets up stack, then calls into C runtime library,
  - Runtime 1st initializes memory and other libraries,
  - then calls your procedure named main ()
- We'll see how to retrieve command-line arguments in main() later…

A Second C Program: Compute Table of Sines

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    int    angle_degree;
    double angle_radian, pi, value;
    /* Print a header */
    printf("\nCompute a table of the
   sine function\n\n");

    /* obtain pi once for all       */
    /* or just use pi = M_PI, where */
    /* M_PI is defined in math.h    */
    pi = 4.0*atan(1.0);
    printf("Value of PI = %f \n\n",
   pi);
```

```
printf("angle     Sine \n");

angle_degree = 0;
/* initial angle value */
/* scan over angle     */
while (angle_degree <= 360)
/* loop until angle_degree > 360 */
   {
      angle_radian = pi*angle_degree/180.0;
      value = sin(angle_radian);
      printf (" %3d      %f \n ",
                angle_degree, value);
      angle_degree = angle_degree + 10;
      /* increment the loop index */
   }
return 0;
}
```

Second C Program Sample Output

```
Compute a table of the sine
   function

Value of PI = 3.141593

angle      Sine
   0       0.000000
  10       0.173648
  20       0.342020
  30       0.500000
  40       0.642788
  50       0.766044
  60       0.866025
  70       0.939693
  80       0.984808
  90       1.000000
 100       0.984808
 110       0.939693
 120       0.866025
 130       0.766044
 140       0.642788
 150       0.500000
 160       0.342020
 170       0.173648
 180       0.000000
 190      -0.173648
 200      -0.342020
 210      -0.500000
 220      -0.642788
 230      -0.766044
 240      -0.866025
 250      -0.939693
 260      -0.984808
 270      -1.000000
 280      -0.984808
 290      -0.939693
 300      -0.866025
 310      -0.766044
 320      -0.642788
 330      -0.500000
 340      -0.342020
 350      -0.173648
 360      -0.000000
```

C Syntax: Variable Declarations

- *All* variable declarations must appear before they are used (e.g., at the beginning of the block)
- A variable may be initialized in its declaration; if not, it holds garbage!
- Examples of declarations:
  - Correct: {

```
int a = 0, b = 10;
...
```

  - Incorrect:

```
for (int i = 0; i < 10; i++)
}
```

Newer C standards are more flexible about this, more later

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs in terms of control flow
  - **`if-else`**

    -
      ```
      if (expression) statement
      ```

    -
      ```
      if (expression) statement1
      else statement2
      ```
  - **`while`**

    -
      ```
      while (expression)
          statement
      ```

    -
      ```
      do
          statement
      while (expression);
      ```

C Syntax : Control Flow (2/2)

- `for`
  - `for (initialize; check; update) statement`
- `switch`

  -
    ```
    switch (expression){
       case const1:     statements
       case const2:     statements
       default:         statements
    }
    ```
  - `break`

C Syntax: True or False

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of *pointer*: more on this later)
  - *No explicit Boolean type*
- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true!

C operators

- arithmetic: +, -, *, /, %
- assignment: =
- augmented assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- equality testing: ==, !=
- subexpression grouping: ( )
- order relations: <, <=, >, >=
- increment and decrement: ++ and --
- member selection: ., ->
- conditional evaluation: ? :

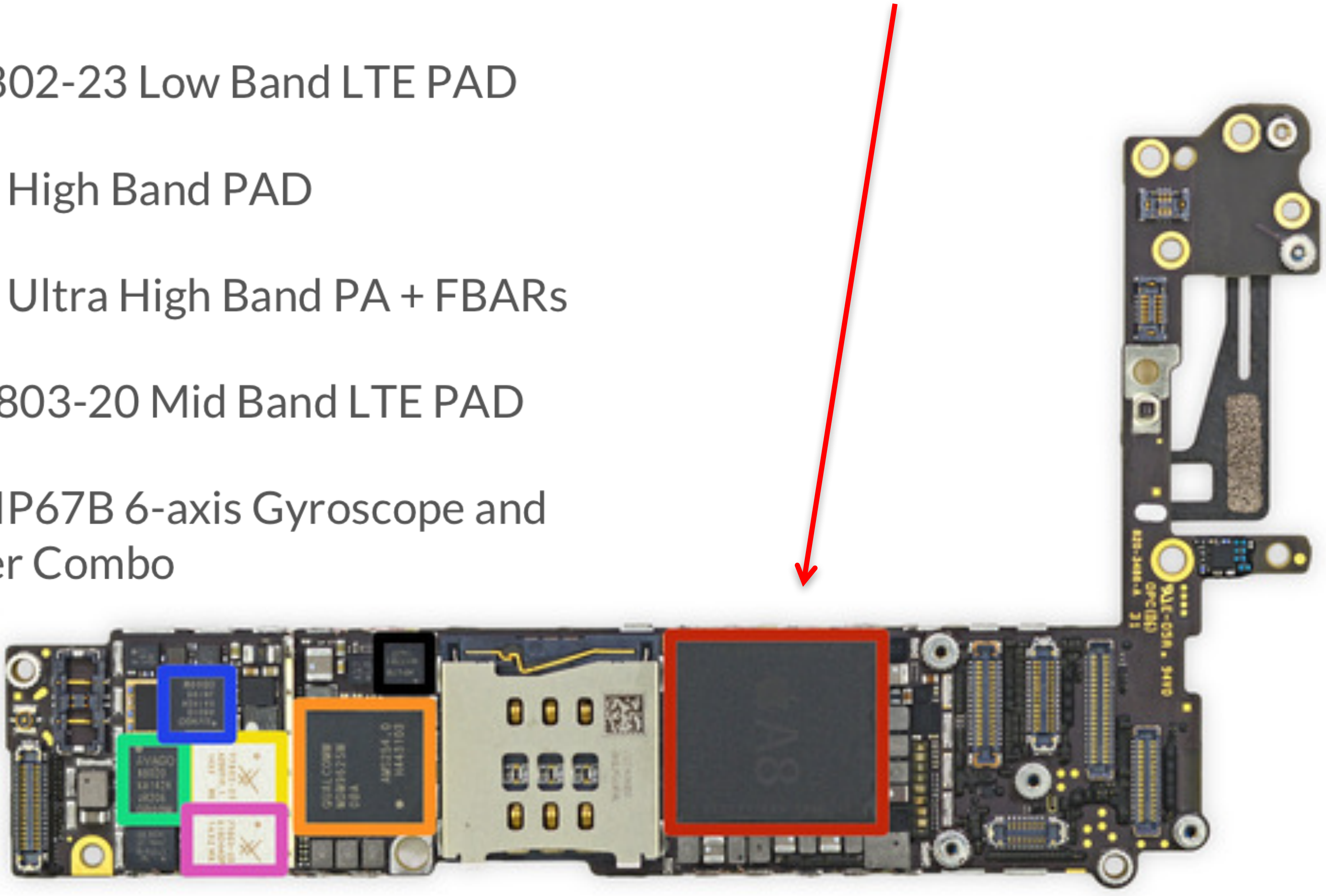
Agenda

- Compile vs. Interpret
- C vs. Java vs. Python
- Administrivia
- Quick Start Introduction to C
- News/Technology Break
- Pointers
- And in Conclusion, …

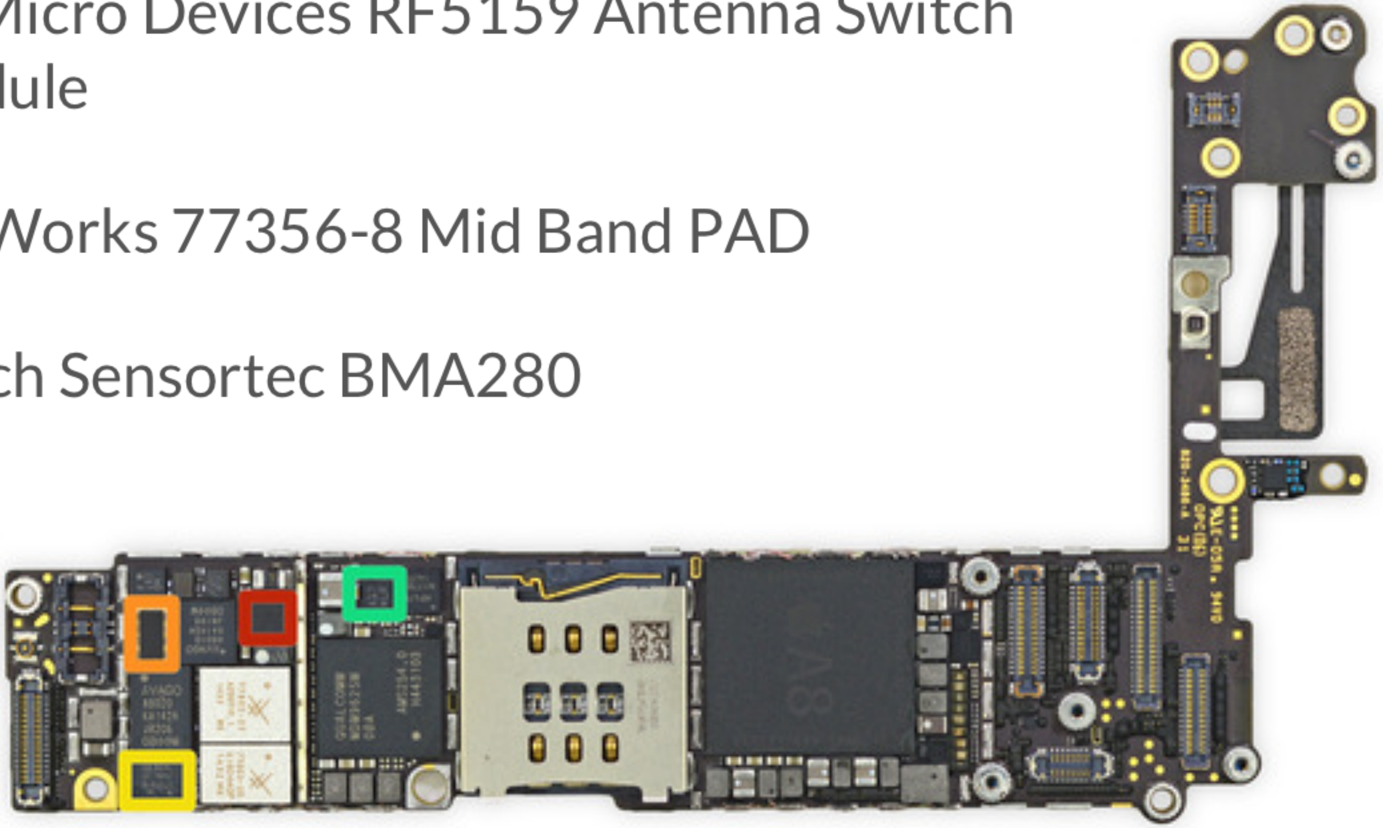
iPhone6 Teardown
fixit.com

- The front side of the logic board:
  - 🔴 Apple A8 APL1011 SoC + SK Hynix RAM as denoted by the markings H9CKNNN8KTMRWR-NTH (we presume it is 1 GB LPDDR3 RAM, the same as in the iPhone 6 Plus)
  - 🟠 Qualcomm MDM9625M LTE Modem
  - 🟡 Skyworks 77802-23 Low Band LTE PAD
  - 🟢 Avago A8020 High Band PAD
  - 🔵 Avago A8010 Ultra High Band PA + FBARs
  - 🟣 SkyWorks 77803-20 Mid Band LTE PAD
  - ⚫ InvenSense MP67B 6-axis Gyroscope and Accelerometer Combo

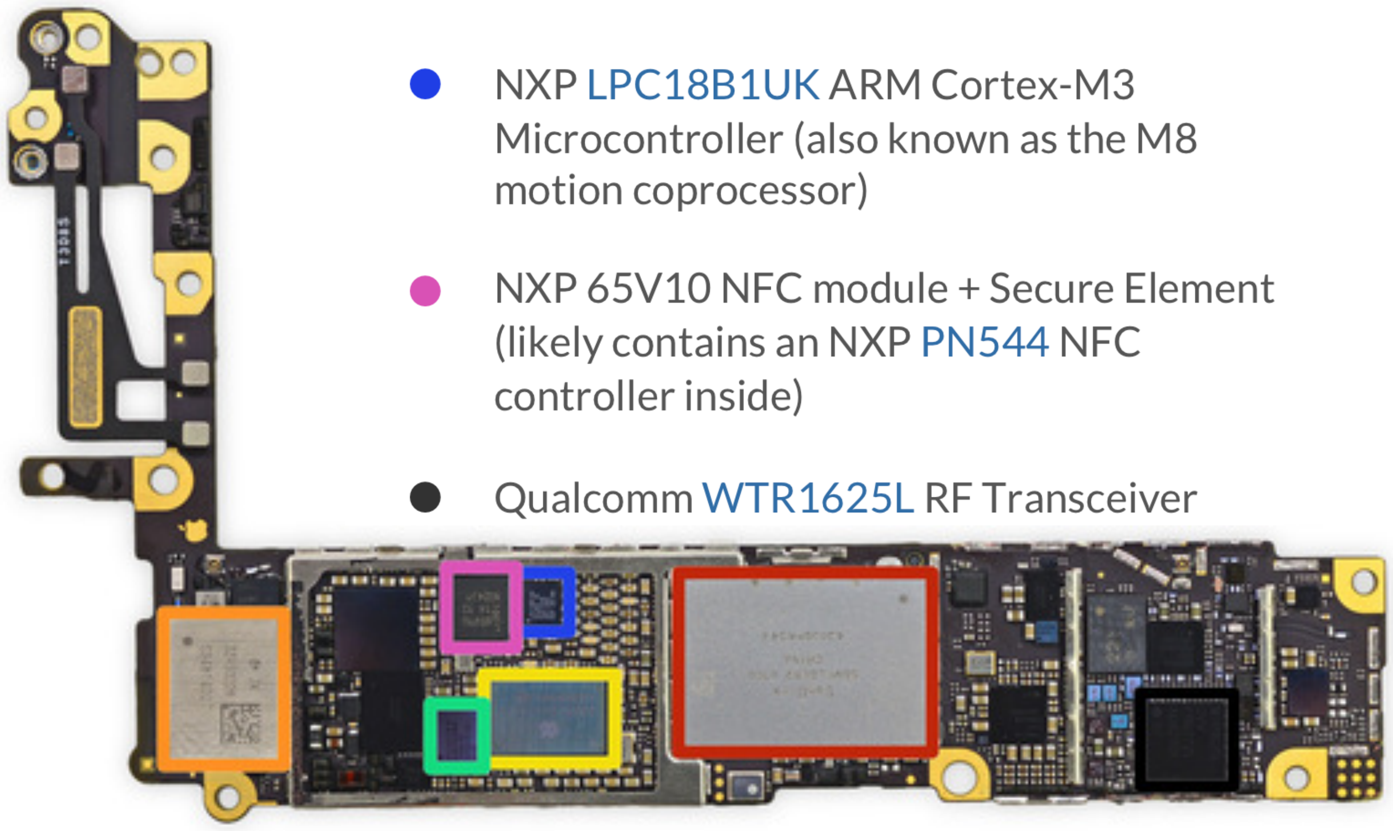
The A8 is manufactured on a 20 nm process by TSMC. It contains 2 billion transistors. Its physical size is 89 mm^2.] It has 1 GB of LPDDR3 RAM included in the package. It is dual core, and has a frequency of 1.38 GHz.



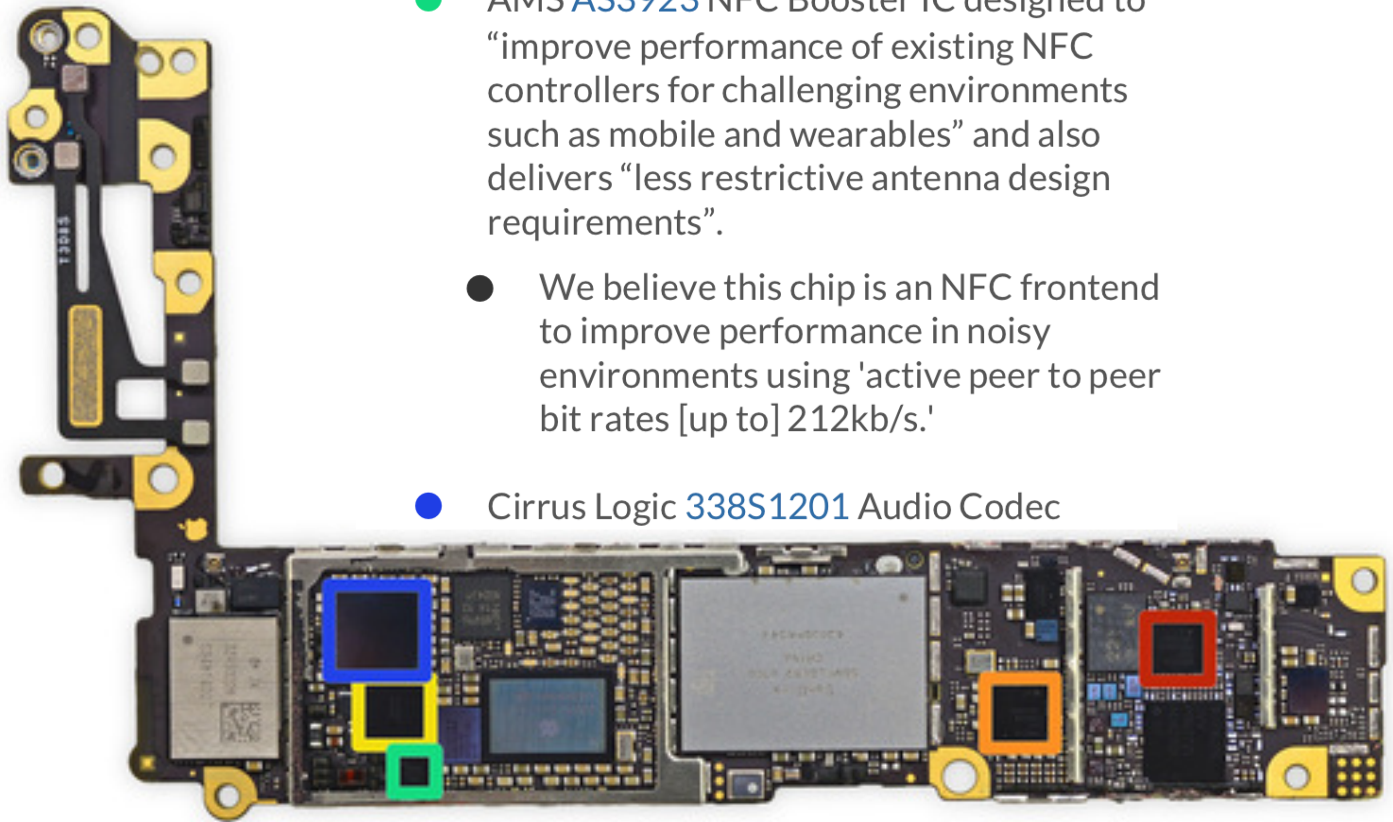
- More ICs on the front side of the logic board:
  - 🟥 Qualcomm QFE1000 Envelope Tracking IC
  - 🟧 RF Micro Devices RF5159 Antenna Switch Module
  - 🟨 SkyWorks 77356-8 Mid Band PAD
  - 🟩 Bosch Sensortec BMA280



- Back side of the logic board:
  - 🟥 SanDisk SDMFLBCB2 128 Gb (16 GB) NAND Flash
  - 🟧 Murata 339S0228 Wi-Fi Module
  - 🟨 Apple/Dialog 338S1251-AZ Power Management IC
  - 🟩 Broadcom BCM5976 Touchscreen Controller
  - 🟦 NXP LPC18B1UK ARM Cortex-M3 Microcontroller (also known as the M8 motion coprocessor)
  - 🟪 NXP 65V10 NFC module + Secure Element (likely contains an NXP PN544 NFC controller inside)
  - ⬛ Qualcomm WTR1625L RF Transceiver



- More ICs await us on the back of the logic board:
  - 🟥 Qualcomm WFR1620 receive-only companion chip. Qualcomm states that the WFR1620 is "required for implementation of carrier aggregation with WTR1625L."
  - 🟧 Qualcomm PM8019 Power Management IC
  - 🟨 Texas Instruments 343S0694 Touch Transmitter
  - 🟩 AMS AS3923 NFC Booster IC designed to “improve performance of existing NFC controllers for challenging environments such as mobile and wearables” and also delivers “less restrictive antenna design requirements”.
    - We believe this chip is an NFC frontend to improve performance in noisy environments using 'active peer to peer bit rates [up to] 212kb/s.'
  - 🟦 Cirrus Logic 338S1201 Audio Codec

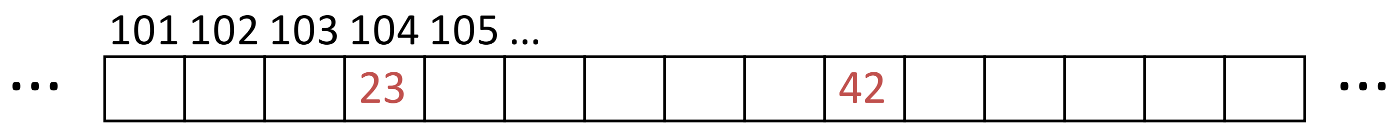


Agenda

- Compile vs. Interpret
- C vs. Java vs. Python
- Administrivia
- Quick Start Introduction to C
- News/Technology Break
- Pointers
- And in Conclusion, …

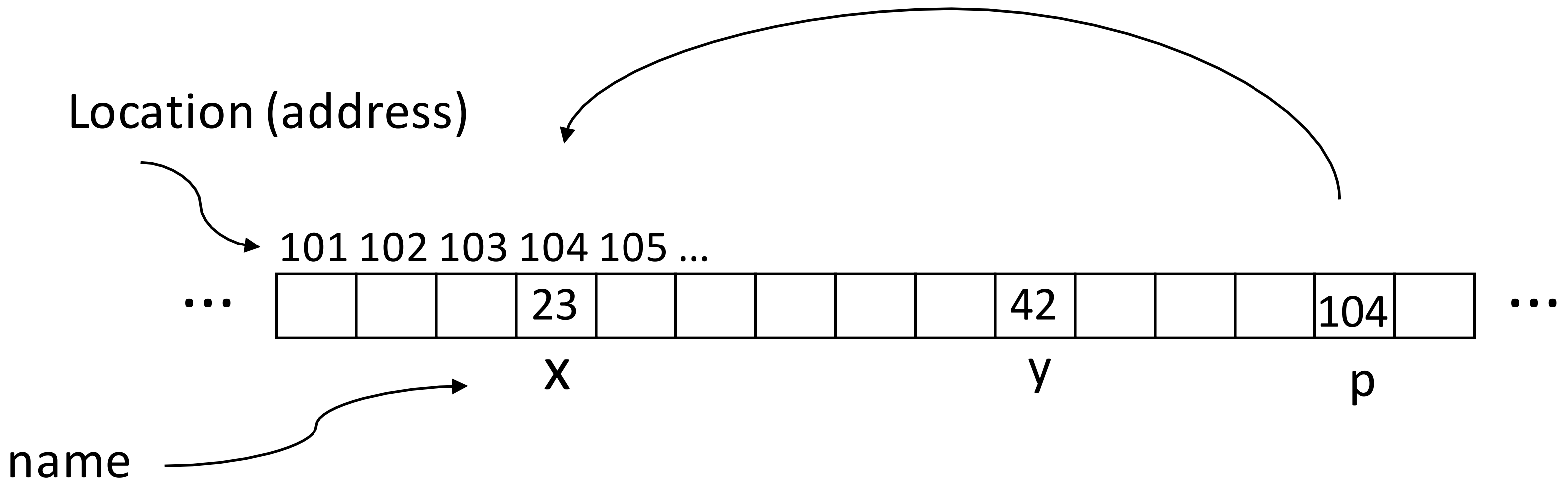
Address vs. Value

- Consider memory to be a single huge array
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - For addresses do we use signed or unsigned numbers? Negative address?!
- Don't confuse the address referring to a memory location with the value stored there



Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable



Pointer Syntax

- `int *x;`
  - Tells compiler that variable x is address of an `int`
- `x = &y;`
  - Tells compiler to assign address of `y` to `x`
  - `&` called the “address operator” in this context
- `z = *x;`
  - Tells compiler to assign value at address in `x` to `z`
  - `*` called the “dereference operator” in this context

Creating and Using Pointers

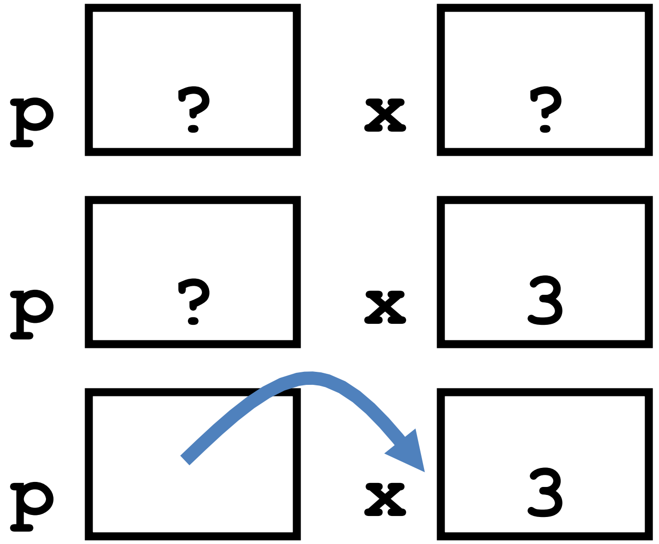
- How to create a pointer:

  `&` operator: get address of a variable

```
int *p, x;

    x = 3;

    p = &x;
```



Note the "*" gets used 2 different ways in this example. In the declaration to indicate that **p** is going to be a pointer, and in the **printf** to get the value pointed to by **p**.

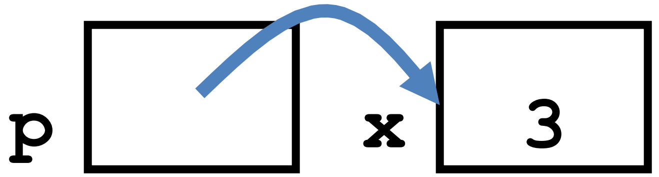
- How get a value pointed to?

  "*" (dereference operator): get the value that the pointer points to

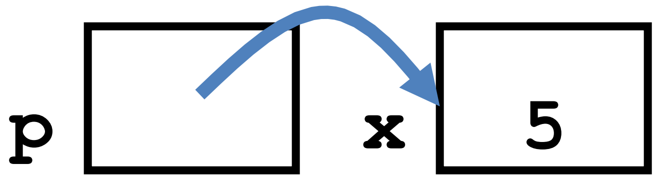
```
printf("p points to %d\n",*p);
```

Using Pointer for Writes

- How to change a variable pointed to?
  - Use the dereference operator **`*`** on left of assignment operator **`=`**



```
*p = 5;
```



Pointers and Parameter Passing

- C passes parameters "by value"
  - Procedure/function/method gets a copy of the parameter, *so changing the copy cannot change the original*

```
void add_one (int x) {
    x = x + 1;
  }
int y = 3;
add_one(y);
```

y remains equal to 3

Pointers and Parameter Passing

- How can we get a function to change the value held in a variable?

```
void add_one (int *p) {
   *p = *p + 1;
  }
int y = 3;

add_one(&y);
```

y is now equal to 4

What would you use in C++?

Call by reference:

```
void add_one (int &p) {
   p = p + 1;    // or  p += 1;
}
```

Types of Pointers

- Pointers are used to point to any kind of data (**`int`**, **`char`**, a **`struct`**, etc.)
- Normally a pointer only points to one type (**`int`**, **`char`**, a **`struct`**, etc.).
  - **`void *`** is a type that can point to anything (generic pointer)
  - Use **`void *`** sparingly to help avoid program bugs, and security issues, and other bad things!

More C Pointer Dangers

- *Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!*
- Local variables in C are not initialized, they may contain anything (aka "garbage")
- What does the following code do?

```
void f()
{
        int *ptr;
        *ptr = 5;
}
```

Pointers and Structures

```
typedef struct {
     int x;
     int y;
} Point;

Point p1;
Point p2;
Point *paddr;
```

```
/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too */
p1 = p2;
```

Pointers in C

- Why use pointers?
  - If we want to pass a large struct or array, it’s easier / faster / etc. to pass a pointer than the whole thing
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    - Most problematic with dynamic memory management—coming up next week
    - *Dangling references* and *memory leaks*

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn't produce efficient code
  - Computers 25,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today's compilers produce much better code, so may not need to use pointers in application code
- Low-level system code still needs low-level access via pointers

Pointer Fun with

Binky

by Nick Parlante

This is document 104 in the Stanford CS Education Library — please see cslibrary.stanford.edu for this video, its associated documents, and other free educational materials.

Carpe Post Meridiem!

Quiz: Pointers

```
void foo(int *x, int *y)
{   int t;
    if ( *x > *y ) { t = *y; *y = *x; *x = t; }
}
int a=3, b=2, c=1;
foo(&a, &b);
foo(&b, &c);
foo(&a, &b);
printf("a=%d b=%d c=%d\n", a, b, c);
```

Result is:

A: **a=3 b=2 c=1**

B: **a=1 b=2 c=3**

C: **a=1 b=3 c=2**

D: **a=3 b=3 c=3**

E: **a=1 b=1 c=1**

C Arrays

- Declaration:

```
int ar[2];
```

declares a 2-element integer array: just a block of memory

```
int ar[] = {795, 635};
```

declares and initializes a 2-element integer array

C Strings

- String in C is just an array of characters

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (aka “null terminator”)

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```

Array Name / Pointer Duality

- *Key Concept*: Array variable is a "pointer" to the first (0th) element
- So, array variables almost identical to pointers
  - **`char *string`** and **`char string[]`** are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
- Consequences:
  - **`ar`** is an array variable, but works like a pointer
  - **`ar[0]`** is the same as **`*ar`**
  - **`ar[2]`** is the same as **`*(ar+2)`**
  - Can use pointer arithmetic to conveniently access arrays

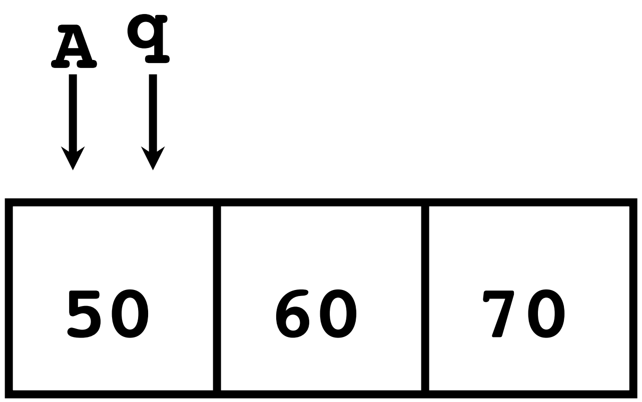
Changing a Pointer Argument?

- What if want function to change a pointer?
- What gets printed?

```
void inc_ptr(int *p)
{    p =  p + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
inc_ptr( q);
printf("*q = %d\n", *q);
```

```
*q = 50
```

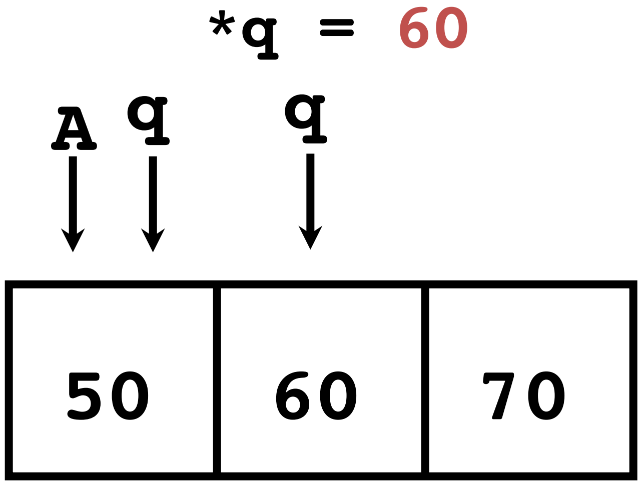


Pointer to a Pointer

- Solution! Pass a pointer to a pointer, declared as **`**h`**
- Now what gets printed?

```
void inc_ptr(int **h)
{    *h = *h + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
inc_ptr(&q);
printf("*q = %d\n", *q);
```



C Arrays are Very Primitive

- An array in C does not know its own length, and its bounds are not checked!
  - Consequence: We can accidentally access off the end of an array
  - Consequence: We must pass the array *and its size* to any procedure that is going to manipulate it
- Segmentation faults and bus errors:
  - These are VERY difficult to find;
    be careful!

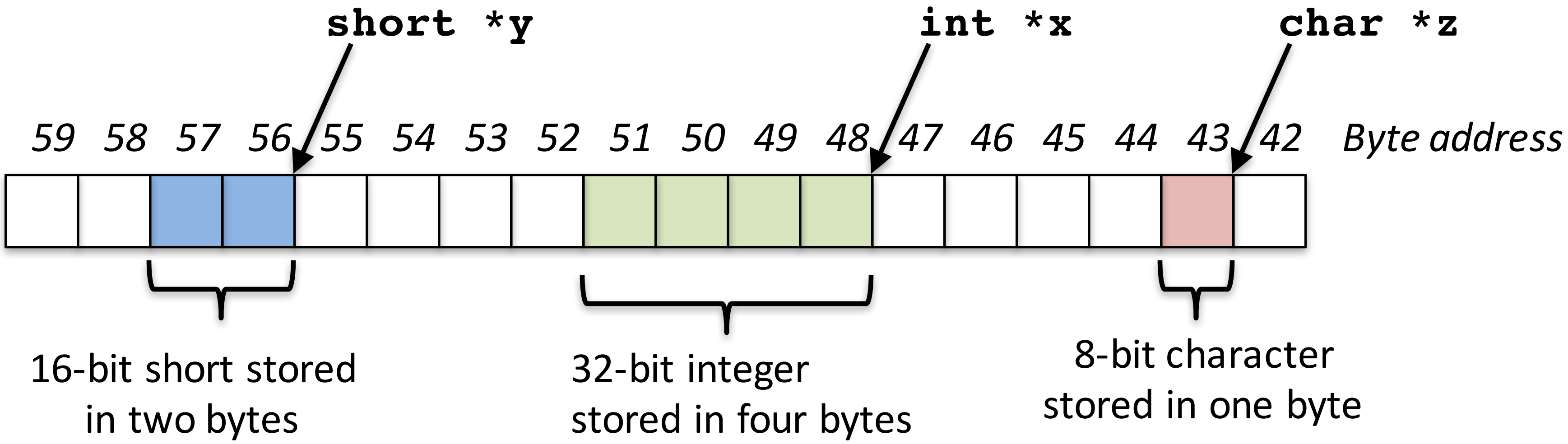
Use Defined Constants

- Array size *n*; want to access from *0* to *n-1*, so you should use counter AND utilize a variable for declaration & incrementation
  - Bad pattern
    ```
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  - Better pattern
    ```
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```

- SINGLE SOURCE OF TRUTH
  - You’re utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: “Don’t Repeat Yourself”

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
  - Hardware's memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes



short *y
int *x
char *z
59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42
Byte address
16-bit short stored in two bytes
32-bit integer stored in four bytes
8-bit character stored in one byte

sizeof() operator

- sizeof(type) returns number of bytes in object
  - But number of bits in a byte is not standardized
    - In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
- By definition, sizeof(char)==1
- Can take sizeof(arr), or sizeof(structtype)
- We'll see more of sizeof when we look at dynamic memory management

Pointer Arithmetic

pointer + number   *pointer – number*

e.g., *pointer* **+ 1**   adds 1 <u>something</u> to a pointer

```
char    *p;
char     a;
char     b;

p = &a;
p += 1;
```

```
int    *p;
int     a;
int     b;

p = &a;
p += 1;
```

In each, p now points to b (Assuming compiler doesn't reorder variables in memory. ***Never code like this!!!!***)

Adds **`1*sizeof(char)`** to the memory address

Adds **`1*sizeof(int)`** to the memory address

Pointer arithmetic should be used <u>cautiously</u>

Arrays and Pointers

- Array ≈ pointer to the initial (0th) array element

`a[i] ≡ *(a+i)`

- An array is passed to a function as a pointer
  - The array size is lost!

- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!

Passing arrays:

Really `int *array`

Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
}
```

Arrays and Pointers

```
int
foo(int array[],
     unsigned int size)
{
    …
    printf("%d\n", sizeof(array));
}

int
main(void)
{
    int a[10], b[5];
    … foo(a, 10)… foo(b, 5) …
    printf("%d\n", sizeof(a));
}
```

What does this print? **8**

… because `array` is really a pointer (and a pointer is architecture dependent, but likely to be 8 on modern machines!)

What does this print? **40**

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These code sequences have the same effect!